Group: Wesley Kim (wesleyk), Nikhil Tibrewal (nibrewa), Jessica Tumarkin (jtumarki)

15-214 Facelook Design Report

Overall Design Model

To maintain a sufficient level of abstraction in communication between different parts of the application, we decided to use the Model-View-Controller (MVC) design pattern in creating our application. This is a large project (we have about 30 java class files in our project). Thus modularity in the code is extremely helpful - it separates the code by functionality, which in turn makes the code much easier to understand. The MVC pattern delegates sections of a process to different parts. This makes understanding the control flow of the application and debugging very easy. The views (our gui package) in our project grab the information in forms, and call the needed method in the specific controller to carry out the action. This way, views don't have to worry about how the action is to be performed on the database level. Similarly, the controller (our facelook package) communicates with the server using a custom protocol, i.e. it doesn't do the database level calls itself. That is left to the 'model' (server and db packages), which is the Server and the DAO files that access the database directly.

Transmitting Data Between Client and Server

We used a JSONWriter to serialize the information to be sent in JSON format and passed it to the server using BufferedReader and PrintWriter. The custom protocols are specific to the particular action that needs to be performed (i.e logging in, adding a friend, etc). The server then just reads in this information from the controller, parses into a JSON object, and uses that information to get the necessary data from the database. For example, in the case where a user is logging in,, the JSONWriter creates an object that contains the email and password that the user enters in the login form. The custom protocol for this sends the server a string with "LOGIN " + <JSON serialized information>. And so we only need to check that information against the database and return a true or false. This kind of return value is easy to pass back to the controller. However, in the case of posts, this is not a trivial task. Posts are returned as arraylists, and since the server can only print out strings, we needed a way to send this information back to the controller. Furthermore, it should be easy enough for the controller to parse it back to an array list too. We again used JSONWriter for this, but now the JSONWriter writes a JSONArray where each JSONObject is a post. This way the JSONTokener straightaway gives a JSONArray and the arraylist of posts can be extracted easily out of it. There is some overhead in converting the array list to a JSONArray, then to a string, and then back to a JSONArray and then to an arraylist. However, this allows our code to be both compact and understandable. Furthermore, given that JSON library methods are stable, this method is less prone to bugs. In fact we did define our own string parsing protocol for passing in subscriptions information between the controller and the model by converting an array list to a string and parsing it using various delimiters such as commas and five dots (.....). But this was too buggy - so we changed that to use JSON objects too.

Our code for controllers and database accessor objects is very modular as well. The server keeps instances of all the necessary database accessor objects, and uses them as necessary. This helps to write methods specific to the part we are concerning with. For example the UserDAO.java class only deals with doing SQL queries to the Users table in the database. So to get all users in the social network for example, one only needs to use the UserDAO and not the PostsDAO. This made debugging a lot easier since we knew exactly which database call failed. Furthermore it keeps the code organized.

Data Organization in Database

Our Database is organized into four tables: Users, Friends, Subscriptions, and Posts. Each table holds information relevant to its namesake. Furthermore, we have some checks within our database to help reduce errors in the system. For example, the email column of the Users table is kept as a unique primary key so there cannot be multiple users with the same email. Thus, in this way we can use the email as the primary username/id. While subscriptions are a one way

relationship, friendships require two way consent. So, the way we organized the Friends table is that the email1 column represents a user who has friended the corresponding user in the email2 column. So, user A and user B are only friends when there are two entries in the Friends table: one with email1 = A and email2 = B, and one with email1 = B and email2 = A. The add/remove functionality is explained in the modifyFriend() method of the FriendsDAO.java file in the db package. The rest of our database is pretty straight forward.

Sharding

We decided to implement sharding by having the user start 5 different servers. As detailed in the README.txt (in the /src directory), these servers are required to be on ports 15210-15214 - so that we know which ports to connect to when storing and reading data from various tables on the servers. Every time we wished to store data - for instance, the login details of a certain user - we would assign it to a certain server based on a hashing of the user's email. This worked out appropriately for all of our databases, since we would store the posts of a certain user, their subscriptions, friends, and login details in a server that was based on the hash of that user's email. An issue did arise, however, when storing the friends of a certain user. Due to the way we stored a user's friends, we would have to ensure that both directions of a friend relationship (email 1 is friends of email 2 and email 2 is friends of email 1) were stored in 2 shards - the ones directly related to email1 and email2. This resolved any issues that we had relating to determining relationships between users.

There are many positive aspects of sharding - it reduces strain that might result on one server by having user data spread out among multiple servers. However, it could still result in strain being placed on one shard if a lot of data that is related to one particular user is repeatedly requested. That's why a cache is so essential - this way, the server won't have to take the strain of handling a lot of requests if popular users' info is placed in a cache instead.

Caching Techniques

We implemented our cache with a built in structure in the Java API: the LinkedHashMap. We extended this structure with our own Cache.java (in the db package). This structure gives the efficiency of look-ups that come with a HashMap, yet it also keeps an internal ordering that Linked Lists have. The internal ordering came in handy when we implemented our eviction policy as the least recently used policy. The structure actually has a removeEldestEntry method that we overwrote in our Cache object. In this method, we specified that if the size of the cache is greater than the initial capacity set, then the cache should remove its oldest entry. Thus, whenever an element is added to the cache, all the eviction is handled internally. This goes directly along with the LRU policy.

For the key for our Cache, we used a combination of the username (email) and whether or not the posts would only be seen to friends (essentially, if it's a status or a notification). The actual values that would be cached are ArrayLists of Posts. Thus, the idea is that when someone visits user A's Profile Page, the set of ten posts that are received are cached, and linked to user A, along with whether the set of posts include both status and notifications, or just notifications. This is just one case where the cache would be utilized.

Although we did not implement the cache as it was exactly supposed to be implemented (each cache stores the data of a specific server shard rather than storing information found across all shards), we still feel that our caching does a lot to reduce strain on a particular shard. This implementation speeds up the retrieval of user information by having it stored in a cache and still reduces load on servers. Due to time constraints, we could not fix our current implementation to make it exactly fit the specifications. For future reference, it would be better to have the specific cache functionality

detailed more clearly in the documentation rather than stating that “Zark is...afraid that the machine which hosts his profile will be overwhelmed with calls for his most recent status updates. To fix this, he wants you to implement a cache so that this won’t be as much of a problem." We interpreted this to mean that each cache could simply store the relevant data of a particular shard to reduce server load rather than storing data from all shards, which seems like it was not the intended functionality. Furthermore, this aspect of the assignment was not included under the “Caching” section of the assignment doc, making it also easy to forget about as we completed the assignment.

Evaluation of Assignment

This assignment is conceptually a wonderful idea. It integrates a huge amount of content that we’ve learned throughout the year, including GUI usage, threading, distributed systems, design patterns, etc. However, we feel that some parts of the assignment were not outlined very clearly in the documentation. Since this assignment is so detailed, it is essential that every part is thoroughly understood when each group begins the assignment. Some of the functionality, like adding and removing friends/subscriptions, was not clearly detailed in either the homework outline or the comments within the code. The caching was also very confusing, and it was unclear to our group what exactly the cache should be storing. We required multiple clarifications on Piazza and during office hours before we fully understood this section of the assignment. However, other parts were done very well - the GUI code was fully functional and for the most part, it was clear where we should have been inserting code and adding our own functionality. For suggestions for the future, perhaps extra functionality like being able to write on a friend’s wall should be included. Additionally, since many people do not have experience using databases, it might be helpful to go over SQLite in detail during class. Finally, it seems like the assignment would have been better suited to have a two-week long span. It’s unfortunate that it ended up being that students could miss out on this assignment with one homework being dropped, as we believe it was a good experience. Rather, in future semesters, it seems like the assignment would work really well in two weeks: the first week implemented the application specifications, and the second week implementing sharding and caching. Overall, our group agrees that this is currently a wonderful assignment, we just wish that it was better organized and documented so that every group could have gotten the most out of it.